TP311.5

用于软件维护的集成生存期模型(续)

Stephen S. Yau, Robin A. Nicholl

Jeffrey J.-P. Tsai, Sying-Syang Liu

七、 集成生存期模型的构造

在这一节，我们讨论如何从软件开发项目不同阶段的描述构造集成生存期模型的各个部分。首先，我们描述构造某一具体阶段模型的方法，再描述构造阶段间模型的方法。在维护开始之前，最好由软件系统的开发者来构造这个模型。构造集成生存期模型的需要与开发软件的方式并无冲突的意思。我们期望使用自动工具，从标准开发资料的机器可读版本来建造各阶段的图模型，建造时使用的技术象构造通用程序模型中所用的技术一样。我们期望在软件开发活动期间建造阶段间模型，作为开发决策的记录。如果这些模型不在开发期间建造，那么必须在维护开始之前建造它们，结果损失了准确的信息。无论在哪种情况下，都需要额外的工作量。我们将描述从两个图模型构造阶段间模型所涉及的工作，而且可能期望自动工具能够帮助用户承担起这个任务。

1. 一个阶段模型的构造

单个阶段的图模型以语义性质而不是语法性质为基础，假定这个基础足够广泛，能够支持此模型的一般目的。为此，对一个具体表示法构造一个阶段模型之前，必须预备好此表示法的语义定义。

例如，让我们考虑Pascal语法造构 ‹compound statement› 的 BNF 产生式规则，它可以定义如下：

‹compound statement› ::=

begin ‹statement› {; ‹statement›} end.

然而，为了语义模型起见，我们感兴趣的

通道㉘：定义一组程序成分的图像，这组成分实现提供者端口与用户端口间的数据传送，这可能像过程或任务入口调用那样简单，也可能像程序包和任务那样复杂**)。

通道样板：定义一个通道类型的图像，它可以用来产生通道实例；它使用和通道相同的图像*)，**)。

通道实例：从通道样板实例化的一个通道，它使用和通道相同的符号*)，**)。

接收口插头㉙：提供者或用户端口的一个调用连接点，接收口插头把一个进入端口的调用和程序包外部过程联系起来**)。

发送口插头㉗：提供者或用户端口的一个调用连接点；发送口插头把一个从端口出去的调用和一个带标号的连接点联系起来**)。

参考文献（共58篇略）

〔示 元译自«IEEE TRANS. ON SOFTWARE ENG.» Vol. 15 No.3 1989. p235—249，王 心校〕

*) 在操作上，从样板实例化的任一任务、程序包或通道不具有任何可存取的内部结构；即所有内部结构都是与原样板相联系的。因此，必须对样板（不是实例）执行实例的内部结构的编辑。

**) 这些图像被包括以支持使用Estelle结构的通信协议软件设计[30]。Ada设计的重要性在于为连接的可视封装提供了机制（Ada不具有连接的程序包概念）。

是，以语句出现的次序而不是使用“begin”，“end”，和“；”作为这一构造的定界符来执行这个语句表。因此，我们可以从这样的构造“抽象出来”，给出以下的产生式规则：

‹compound statement› ::=
‹statement› {→ ‹statement›}

其中$S_1 \rightarrow S_2$表示在S_2之前应执行S_1。

类似地，我们希望显示数据流和数据结构性质。数据流性质可能与下面方式的Pascal赋值语句有联系，这些语句使用属性I和O来分别表示变量至语句的输入和从语句的输出。

‹assignment statement› ::=
‹variable› : = ‹expression›
‹variable› ::=
‹identifier› | ‹identifier› [‹indexlist›]
‹indexlist› ::= ‹index› {, ‹index›}
‹index› ::= ‹expression›

AS::=I→O (I=V.I∪E.I, O=V.O∪E.O)

V.I::=IL.I和V.O::=id∪IL.O

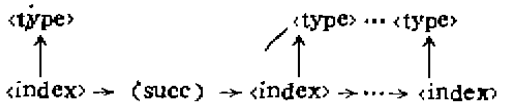
IL.I::=$\cup_i$Ii.I和IL.O::=$\cup_i$Ii.O

I.I::=E.I和I.O::=E.O.

数据结构性质来自被说明的对象结构。例如，在Pascal中，一个数组结构具有如下的表示：

‹array declaration› ::=
array[‹index range list›]of ‹type›
‹index range list› ::=
‹index range› {, ‹index range›}

具有这样的解释：



因此，一个具体表示法在一个阶段模型的定义主要是一个手工过程。必须分析这种表示法以便弄清：确定事件次序的特征；确表信息流的特征；确定信息形式的特征。

表示法的这些特征必须借助模型的基本元素来表征。下面是此模型的一个过程：

·对于语言定义中对应于不同活动的每一构造，定义一个实体。

·对于语言定义中定义其它构造间的排序关系的每一构造，定义一个关系实体。

·对于语言定义中定义流入或流出一个活动的信息流的每一构造，定义一个三元组‹activity, I-set, O-set›。

·对于语言定义中定义一块信息形式的每一构造，定义一个‹name, structure›对，其中‹structure›是从信息形式导出的。

作为一个例子，我们描述RSL语言[8]的一个子集用于软件要求的一个模型构造。在下面的定义中，用“‹”和“›”定界非终极符号，用“[”和“]”定界任选符号，用“|”定界在符号间所做的选择，用“{”和“}”定界重复的符号并且在“{”和“}”的前面和后面有下标整数，分别表示迭代次数的下限和上限。现在可以给出此语言子集的定义如下：

‹new element definition› ::=
[DEFINE] element-type-name element-name
[comment]
$_0${[INSERT] ‹element definition sentence›}$_1$,
‹element definition sentence› ::=
‹attribute declaration›
| ‹relation declaration›
| ‹structure declaration›
‹attribute declaration› ::=
attribute-name
$_1$(value-name | number | text-string)$_1$
[comment].
‹relation declaration› ::=
relation-name[relation-optional-word]
$_1$([element-type-name]element-name
[comment])$_n$.
‹structure declaration› ::=
STRUCTURE$_1${ ‹node› }$_n$END[comment].

```
‹node› ::=
  ‹element node›
  | ‹terminator›
  | ‹and node›
  | ‹or node›
  | ‹for-each node›
‹element node› ::=[element-type-name]
  element-name[comment]
‹terminator› ::=
  TERMINATE[comment]
  |RETURN[comment]
‹and node› ::=
  DO[comment]‹branch›
    1{AND ‹branch›}n
  END
‹branch› ::=1{‹node›}n
‹or node› ::=
  IF[comment]‹conditional branch›
    n{OR ‹conditional branch›}n
  OTHERWISE[‹branch›]
  END
‹conditional branch› ::=
  [unsigned-integer]‹condition› ‹branch›
‹for-each node› ::=
  FOR EACH[FILE]file-name[RECORD]
  [SUCH THAT ‹condition›]
  DO[comment]
    1{[ALPHA]alpha-name[comment]
      |[SUBNET]subnet-name[comment]}1
  END
‹condition› ::= (‹Boolean expression›)
```

效仿以上的定义过程，现在我们可以确立以下的过程步骤：

1）对软件系统要求中的每个ALPHA，SUBNET和STRUCTURE构造一个实体。

2）对每个‹and node›，‹terminator›，‹or node›和‹for-each node›构造一些关系实体。

3）对每个ALPHA，SUBNET和STRUCTURE定义三元组‹alpha-name，I-set，O-set›，‹subnet-name，I-set，O-set›，和‹structure-name，I-set，O-set›。

4）根据DATA项定义，定义‹data item name，structure›对。

2. 阶段间模型的构造

阶段间模型的构造应在软件维护开始之前，最好由软件开发者来进行，否则必须由修改系统的软件维护人员进行修改。假如给定了软件系统的一个源模型和从其导出的一个目标模型，就可以根据下面列出的基本过程，构造这两个阶段内模型的阶段间模型。帮助这一任务的自动工具应侧重于用户资料中出现的概念，而不是图模型中出现的概念。因此，下面构造阶段间模型的过程细节应向建造模型的人隐蔽起来。

1）对源模型中的每个结点，将其分配给一规则的左端。

2）对每条规则，把目标模型的一个子图分配给其右端。

3）对作为子图与全图模型间接口一部分的左端每一结点，向用户指明该结点可以连接到模型其余部份的任何弧。

4）根据用户的应答，选择下列步骤之一：

a)如果弧不出现在下一个阶段，那么开发者必须输入关于这一省略的解释。

b)如果弧在下一阶段给以表示，那么开发者必须弄清结点或那些"表示"考虑中结点的右端结点。右端每个这样的结点应给以一个唯一的标记。左端的结点应给以一个标记表，它由正好分配给右端的所有标记组成。

c)如果在打算遵从最后一步的说明时，发现弧不是由下一阶段中某一弧或某一组弧表示的，那么我们应把这条弧增加到左端，判定左端是否可以分成更简单的子图，并且以相应的方式修改右端。

八、 集成生存期模型举例(略)

九、 在软件修改中的应用

在这一节，我们描述如何应用我们的模型来识别根据更改请求需要修改软件系统的所有项目。我们首先描述在一个阶段中如何弄清这些项目，然后描述如何弄清几个阶段中的这些项目。由于软件系统的所有增强起源于用户级，所以我们假定识别起初任何改

变的地方得由用户完成。

跟踪过程将说明当考察可能的改动带来的影响时，必须对图和重写规则进行何种形式的分析。用户有可能不希望使用这些面向图的概念，与自动工具打交道。自动工具应完成这些过程，而同时允许用户借助软件资料中出现的一些概念进行交互。这样一些技术已包括在，例如面向语言的编辑器中，这些编辑器允许借助程序文本表达用户交互，而且可以表示由语法分析树正在编辑的程序。

1. 阶段内跟踪

我们感兴趣的是，跟踪对系统的这一阶段所做的改动将对系统的其它部分产生的影响。这些问题十分类似于程序修改问题，后者已经提出了部份解决办法[4-7]。我们的跟踪方法基本上是对现有的逻辑和性能波动效应分析技术的扩充[6,7]，这些技术使用了逻辑和性能相关性的图模型来跟踪程序的改动产生的影响。经过拓广这些技术，我们可以跟踪模型中表示的与维护程序员认为适宜的任何性质。这种方法直接以图为基础，这种图表示：如果在一次修改中已牵涉到图模型的任一结点N，那么与N邻接的任一结点也被认为受到牵联。被认为受到牵联的初始结点集就是尚待修改的结点集。由这个初始集所直接牵涉的结点包括在尚未修改的系统中邻接的结点和由于修改的结果变成的邻接结点。

2. 阶段间跟踪

我们首先举一个抽象的例子，说明我们的阶段间跟踪的方法，然后继续定义用于"本原"修改活动各种类型的过程。我们将说明，使用图9 (b) 的跟踪规则，如何跟踪改动图9 (a) 中描述的"系统"所产生的影响到图9 (c) 中描述的下一级系统分解。注意，在图9 (b) 给出的规则中，系统的每一个结点出现在恰好一条规则的左端。现在我们考虑对原系统结构所做的修改影响。

让我们考虑标记为C的结点的修改。这个结点似乎完全局部化在原始图中。因此，这一修改的波动效应不应太大。

1) 为了进行局部波动效应分析，我们得考察与C邻接的结点B和E。假如B和E不需要改变，这就是阶段内的跟踪步骤。

2) 跟踪规则表明，标记为C的结点只出现在R4中，其中对子图RHS (R4) 正跟踪结点C。

3) 现在我们必须确定RHS (R4) 的哪些部分必须改变。让我们假定，我们决定用图9 (d) 中所示的RHS′ (R4) 来代替它。我们可以认为这是增添一个新的特性Y以及修改一个现有特性V。由于插入标记为Y的结点和修改标记为V的结点为标记为V′的结点，所以现在我们必须进行RHS′ (R4) 的一些局部波动效应分析。然而，我们期望，一个规则的右端应足够小，使得这个分析透彻，也许顺手。

4) 现在，不必进一步对修改后的重写系统做波动效应分析，因为标记为X的结点是该规则中唯一的粘合项并且它没受到牵联。因此，如果（在上面步骤3中）已经验证了它，那么不可能进一步发生波动效应。

5) 跟踪这些效应到下一级。在这种情形下，可以用在前一级修改标记为C的结点的同一方式处理对标记为V的结点做的修改。然而，插入标记为Y的结点，必须进行不同的处理，因为举个例说，对新的结点Y还尚未建立规则。

对于新插入的结点，应遵守以下的过程：

1) 对新结点的各个邻接结点，确定其规则是否应包括新结点。如果应包括新结点，那么修改规则的左端，并且对于结点的修改，可以如前面那样进行。否则，继续下一步。

2) 定义一条新的阶段间跟踪规则，其左端是新结点，其右端是新结点的语义定义求精。

3) 确定右端应如何装入新的一级。（嵌入程序）。

4) 在新的一级执行阶段内分析，确保新的右端"装入"该级的现有图中。若需要的话，进行新的修改。

5) 在下一级重复插入和修改结点的过程。

对于被删除的结点，应遵循下面的过

程。

1)让我们假定RHS′（R4）如图9（c）所示。再次，我们应对这个图进行某些局部"波动效应"分析。在这种情况下，最多检查两个结点和一条弧。

2)除上述以外，由于标记为X的结点(粘合项)不受影响，所以不进行进一步波动效应分析，并且这一级没有必要进一步分析。

3)跟踪波动效应到下一级。

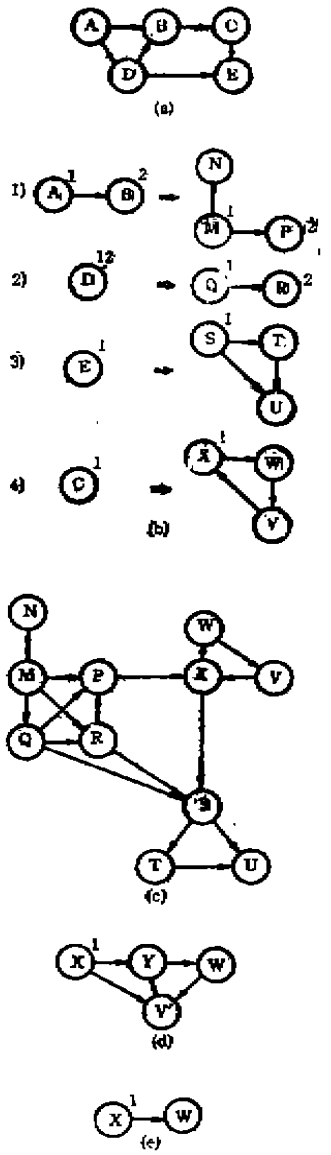


图9 (a)表示一个软件系统的抽象图，(b)一个抽象软件系统的两个阶段间的跟踪规则，(c)抽象软件系统的下一个阶段，(d)产生式规则R4第一个右端方案，(e)规则R4的第二个右端方案。

现在，在下一级，我们首先必须处理涉及标号为V的删除结点的跟踪规则。如果涉及V的规则是一个结点替换规则，那么该规则不再需要了，可以删去。在实现中，我们可能需要保留"不用单元收集"规则的方案，因为这种规则不再能应用。在这一点上，实现的策略取决于我们是否静态地存储描述，或存储跟踪规则，并允许动态地生成描述。

十、讨论及未来的工作

我们使用图重写系统，已经建立了一个集成生存期模型，它能用来寻找和跟踪软件修改在不同阶段产生的影响。最初做这样抉择是因为有许多面向图形的语言用于规格说明和设计，也是因为图可以以"自然"的方式表示基本性质。除此之外，图重写系统的强有力的技术可以精巧地表示阶段间的关系。这一方法的直接好处是可以容易地检索信息。

如果集成模型是为了满足其目标，那么它既不局限于一个具体的表示法，也不局限于任何具体阶段。然而，集成模型可以实现这种通用性，只需要考虑在一个软件系统的元素间存在的可能关系的一个子集。在这一方面进行更多的研究是必要的。首先，我们必须扩充各阶段的模型，以便描述基本关系类型之外的新的类型和关系。其次，我们必须使从各现有阶段到对应图表示的变换和从我们模型的每个阶段到下一个阶段的图表示实现自动化，并且能把这种方法应用到现实世界中的软件项目。为了在任何现有的开发方法中使用我们的模型，我们必须进一步研究变换的复杂性。在开发这种软件工具时所作的一个主要决策是应提出什么形式的用户接口。用户应直接操纵通用图模型，还是用户只应注意共用软件开发资料？无论哪种方法都有可能，因此，在偏向一种方法之前，需要经验。

我们还需要进一步研究，确定这里所给

分布式专家系统协同问解求解

施鸿宝　李　波

（西安交通大学计算机科学与工程系）

摘要

The cooperative problem solving in distributed expert systems which are the combination of expert systems with distributed systems is an attractive branch of expert systems. In this paper we discuss its birth-background, defination and significance, and analysize in detail its cooperative procedure and cooperative strategy. At the same time we classify the problems on the basis of structural feature, and present the idea and architecture for cooperative solving well-structured problems and ill-structured problem in distributed expert systems. At last micro-aspect of cooperative problem solving is described.

一　引　言

1　DES出现的必然性

专家系统（ES）以其固有的理论研究价值在AI和计算机科学中占着重要地位，它使一类新问题可由计算机求解。同时ES以其高性能（在专门领域达到专家水平）和实用性得到了广泛的应用，许多工业和政府都在使用这种技术，新的商业已兴起。目前ES已受到世界各国的普遍关注。

随着ES应用的普及和深入，人们已揭示了导致第一代ES失败的主要原因：

（1）知识不足。由于ES的知识只限于很窄的专业领域，并且可能不完备，对边缘问题其性能变得很坏或根本不能给出有益结果，这就是steels讨论的脆弱性。

（2）方法不妥。McDermott认为第一代ES仅使用一种问题求解方法，已证明很难找到通用方法、通用性与效率成反比。因此各求解方法均针对一定问题而言，不适合其它问题。

事实上，人类专家在自己的专业领域也有许多他不能解决的问题，存在类似的知识不足与方法不妥。但人类专家间的合作就能

出的模型是否适合于其它类型的软件系统和其它软件开发范例。至今，这个模型还没准备表示分布式软件系统，尽管它可以使用模型所允许的非确定性来表示某些形式的并发行为。我们尚不清楚，为支持分布式软件系统的维护，是否必须做一些改变。我们也可能研究我们的集成软件生存期模型和软件开发的变换观点间的联系。看来，我们的阶段间模型的重写规则可以用作从其规格说明建造软件系统的变换规则。这种差别在于，我们的规则是即席构造的，以便描述由软件开发者所做的决策，并且不是来源于需要进行软件开发的某一批标准变换。但我们尚不明白，这种差别有多大。

参考文献（略）

〔仲　源译自《IEEE Trans. Software. Eng.》No.8, Vol.14, 1988. p. 1128-1144, 声　钟校〕